Computer Physics Communications 265 (2021) 108007

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

BEC2HPC: A HPC spectral solver for nonlinear Schrödinger and rotating Gross-Pitaevskii equations. Stationary states computation ☆,☆☆

Jérémie Gaidamour [a], Qinglin Tang [b], Xavier Antoine [a,*]

[a] Université de Lorraine, CNRS, Inria, IECL, F-54000 Nancy, France
[b] School of Mathematics, State Key Laboratory of Hydraulics and Mountain River Engineering, Sichuan University, Chengdu 610064, China

ARTICLE INFO

Article history:
Received 8 December 2020
Received in revised form 9 April 2021
Accepted 16 April 2021
Available online xxxx

Keywords:
Bose-Einstein condensation
Nonlinear Schrödinger equation
Gross-Pitaevskii equation
Stationary states
Pseudo-spectral method
Nonlinear conjugate gradient
High performance computing

ABSTRACT

We present BEC2HPC which is a parallel HPC spectral solver for computing the ground states of the nonlinear Schrödinger equation and the Gross-Pitaevskii equation (GPE) modeling rotating Bose-Einstein condensates (BEC). Considering a standard pseudo-spectral discretization based on Fast Fourier Transforms (FFTs), the method consists in finding the numerical solution of the energy functional minimization problem under normalization constraint by using a preconditioned nonlinear conjugate gradient method. We present some numerical simulations and scalability results for the 2D and 3D problems to obtain the stationary states of BEC with fast rotation and large nonlinearities. The code takes advantage of existing HPC libraries and can itself be leveraged to implement other numerical methods like e.g. for the dynamics of BECs.

Program summary

Program title: BEC2HPC
CPC Library link to program files: https://doi.org/10.17632/mdzpw4dr4t.1
Licensing provisions: GPLv2
Programming language: C++, Python
Nature of problem: This software computes the stationary states of rotating Bose–Einstein condensates (BEC) modeled by the Gross–Pitaevskii equation (GPE). It implements a numerical method that is particularly effective for BEC with fast rotation and large nonlinearities. The parallel implementation allows to perform large-scale simulations of 2D or 3D problems on parallel computing platforms.
Solution method: The stationary states are computed using an iterative pseudo-spectral method based on Fast Fourier Transforms. The computation takes the form of a constrained minimization problem solved using a preconditioned nonlinear conjugate gradient method. This solver is implemented in distributed memory using MPI and a decomposition of the computational domain.
Additional comments including restrictions and unusual features: The algorithms are implemented in C++ and MPI but a Python interface is provided for defining the physics of the problem. Results can be exported to HDF5 files and visualized with external tools such as ParaView. The code can be used to implement other spectral methods in parallel or to solve problems related to the dynamics of BECs.

1. Introduction

Bose-Einstein Condensates (BECs) have been first predicted theoretically by S.N. Bose and A. Einstein, before their experimental realization in 1995 [1–4]. This state of matter has the interesting feature that macroscopic quantum physics properties can emerge and be observed in a laboratory experiment. The literature on BECs grown very fast over the last two decades in atomic, molecular, optics and condensed matter physics. Important applications related to this new physics are now appearing, like e.g. in quantum

☆ The review of this paper was arranged by Prof. N.S. Scott.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail addresses: jeremie.gaidamour@univ-lorraine.fr (J. Gaidamour), qinglin_tang@163.com (Q. Tang), xavier.antoine@univ-lorraine.fr (X. Antoine).
URL: http://iecl.univ-lorraine.fr/~xantoine/ (X. Antoine).

https://doi.org/10.1016/j.cpc.2021.108007
0010-4655/

computing [5]. Concerning the most important directions, a special attention has been directed towards the understanding of the nucleation of vortices [6–12], properties of dipolar gases [13,14] or also multi-components BECs [13,15,16]. For temperatures T which are much smaller than the critical temperature T_c, the macroscopic behavior of a BEC can be correctly described by a condensate wave function ψ as the solution to a Gross-Pitaevskii Equation (GPE). Therefore, being able to numerically calculate efficiently the solution of such a class of equations is extremely useful. Concerning the main questions related to BECs, we can cite the calculation of the stationary states (ground/excited states) as well as the real-time dynamics [13,17–20].

In the present paper, we consider the problem of the computation of stationary states (most particularly ground states) of the rotating GPE. A few methods are available in the literature to numerically obtain them. For example, some algorithms are based on appropriate discretizations of the continuous normalized gradient flow/imaginary-time formulation [13,18,21–27], leading to various iterative algorithms. The methods are general and can be directly extended to many physical situations (e.g. dipolar interactions, multi-components GPEs...) [13,18,22,23,28]. Other approaches rather solve numerically the nonlinear eigenvalue problem [29,30] or the minimization of the energy functional by using optimization techniques under constraints [31–35]. Regularized Newton-type methods can also be used [36]. In the present paper, we consider the constrained nonlinear Preconditioned Conjugate Gradient (PCG) method with a pseudo-spectral discretization scheme for the rotating GPE. This approach, developed in [37] and presented shortly in Section 2, provides an efficient and robust way to solve the minimization problem. Even if only the rotating GPE is considered here, the method can be extended to other situations (see [38] for dipole-dipole interactions or [39,40] for nonlinear fractional GPEs). The main goal of the paper is to introduce the solver BEC2HPC, to explain how to use its functionalities and to provide a few examples. Some future developments will include general systems of GPEs, as well as additional numerical methods for the dynamics of the GPE. Finally, let us remark that the GPE can be simplified to the nonlinear Schrödinger equation. Therefore, the BEC2HPC solver can also be useful to simulate other physical situations than BEC, where the Schrödinger equation has to be numerically solved efficiently and accurately.

Concerning the available solvers for the GPE (and nonlinear Schrödinger equation), several contributions exist in the literature. Adhikari, Muruganandam and their co-authors developed in a series of papers some finite-difference codes for computing the stationary states and dynamics of GPEs without rotation. Various implementations are available, including codes written in Fortran [41,42], C [41,43], with OpenMP/MPI/CUDA/GNU versions [44–46]. In addition, the contribution [47] proposes a code that can solve the GPE with rotation term. In [48], a Matlab toolbox called OCTBEC is designed for the optimal control of BECs. The GPU-accelerated Matlab/C toolbox NLSEmagic is developed in [49] for solving the multi-dimensional nonlinear Schrödinger equation through finite-difference in space and with a fourth-order Runge-Kutta scheme in time. In [50], a finite element C++ toolbox is proposed for computing the stationary states (based on a Newton method) and dynamics of nonlinear Schrödinger equations. Another finite element toolbox has been recently developed with Freefem++ for solving various problems related to the 2D and 3D rotating GPE [51]. Finally, GPELab [28,52] is a Matlab toolbox that can solve a large class of problems related to the GPE, including stationary states, dynamics, and the possibility to handle some additional stochastic terms. The wide variety of problems which can be solved by GPELab leads to the possibility of simulating many complex physical configurations related to BECs, which makes the toolbox very attractive. For the stationary states, the solver used in GPELab may sometimes converge slowly for large nonlinearities and high rotation speeds, since i) it is based on the normalized gradient flow/imaginary time, and ii) the implementation in Matlab does not permit to use optimally the HPC ressources. In BEC2HPC, we use the PCG algorithm which is known to outperfom the normalized gradient flow formulation as shown in [37]. In addition, BEC2HPC proposes a HPC implementation of the algorithm, for the 2D and 3D GPE with rotation term. The resulting solver is then very robust, efficient and highly accurate since it uses a pseudo-spectral approximation in space.

The plan of the paper is the following. In Section 2, we introduce the PCG algorithm that is used in BEC2HPC for computing the stationary states. Section 3 gives some information about the way BEC2HPC is developed, most particularly regarding the FFT implementation and some parallelization aspects. In Section 4, we propose a relatively simple example to start with the use of BEC2HPC, after its installation. Section 5 is devoted to the most advanced features of BEC2HPC, including the data definition and the parameter selection, the use of the visualization tool (ParaView). In Section 6, we provide more examples and report some performances of the solver in 2D and 3D. Finally, we conclude in Section 7.

2. The PCG method for computing stationary states of the GPE

2.1. Notations and formulation

Let us consider the problem of computing a ground state of a d-dimensional ($d = 2, 3$) BEC which can be written under the form of the constrained minimization problem

$$\begin{cases} \text{Find } \phi \in L^2(\mathbb{R}^d) \text{ such that} \\ \phi \in \arg\min_{||\phi||_2=1} \mathcal{E}_{\text{tot}}(\phi), \end{cases} \tag{2.1}$$

where the $L^2(\mathbb{R}^d)$-norm of ϕ is defined as

$$||\phi||_2^2 = \int_{\mathbb{R}^d} |\phi|^2 d\mathbf{x} := <\phi, \phi>$$

and the hermitian inner-product is

$$\forall (u, v) \in L^2(\mathbb{R}^d) \times L^2(\mathbb{R}^d), \quad <u, v> := \int_{\mathbb{R}^d} u v^* d\mathbf{x},$$

defining v^* as the complex conjugate of v. For the minimization problem (2.1), we introduce the total energy functional $\mathcal{E}_{\text{tot}}$ for the dimensionless rotating GPE defined by

$$\begin{aligned} \mathcal{E}_{\text{tot}}(\phi) &= \int_{\mathbb{R}^d} \left[\frac{1}{2} |\nabla \phi|^2 + V(\mathbf{x}) |\phi|^2 + F(|\phi|^2) - \text{Re}(\phi^* \mathbf{\Omega} \cdot \mathbf{L}\phi) \right] d\mathbf{x} \\ &:= \mathcal{E}_{\text{kin}}(\phi) + \mathcal{E}_{\text{pot}}(\phi) + \mathcal{E}_{\text{int}}(\phi) + \mathcal{E}_{\text{rot}}(\phi), \end{aligned}$$

for $t > 0$. In 3D, the Laplace operator is given by: $\Delta = \nabla^2$, where $\nabla := (\partial_x, \partial_y, \partial_z)^t$ is the gradient operator; the spatial variable is $\mathbf{x} = (x, y, z)^t \in \mathbb{R}^3$ (in 2D, we have $\nabla := (\partial_x, \partial_y)^t$ and $\mathbf{x} = (x, y)^t \in \mathbb{R}^2$). The function V represents the external (usually confining) potential. The smooth real-valued function $f(\rho) := F'(\rho)$ models the nonlinearity, setting $\rho = |\phi|^2$ as the density function. A first example consists in the standard cubic case which reads as

$$F(\rho) = \beta \rho^2 / 2, \tag{2.2}$$

and then $f(\rho) = \beta \rho$, where β is the nonlinearity strength describing the interaction between atoms of the condensate. This parameter is related to the s-scattering length (a_s) and is positive (respectively negative) for a repulsive (respectively attractive)

interaction. Other kinds of nonlinearities involve e.g. nonlocal nonlinear interactions like the dipole-dipole interaction [18,38–40,53]. For vortices creation, a rotating term is added. The vector $\mathbf{\Omega}$ is the angular velocity vector and the angular momentum is $\mathbf{L} = (L_x, L_y, L_z) = \mathbf{x} \wedge \mathbf{P}$, with the momentum operator $\mathbf{P} = -i\nabla$. In many situations, the angular velocity is such that $\mathbf{\Omega} = (0, 0, \omega)^t$ leading to

$$\mathbf{\Omega} \cdot \mathbf{L} = \omega \mathcal{L}_z = -i\omega(x\partial_y - y\partial_x). \tag{2.3}$$

A direct computation of the energy gradient yields

$$\nabla \mathcal{E}_{\text{tot}}(\phi) = 2\mathcal{H}_\phi \phi, \quad \text{with } \mathcal{H}_\phi = -\frac{1}{2}\Delta + V + f(|\phi|^2) - \mathbf{\Omega} \cdot \mathbf{L}$$

and the second-order derivative is

$$\frac{1}{2}\nabla^2 \mathcal{E}_{\text{tot}}(\phi)[u, u] = \langle u, \mathcal{H}_\phi u \rangle + \text{Re}\left\langle f(\phi^2), u^2 \right\rangle.$$

Let us introduce $\mathcal{S} = \{\phi \in L^2(\mathbb{R}^d), ||\phi||_2 = 1\}$ as the unit spherical manifold associated to the normalization constraint. The tangent space at a point $\phi \in \mathcal{S}$ is given by $T_\phi \mathcal{S} = \{h \in L^2(\mathbb{R}^d), \text{Re}\langle \phi, h \rangle = 0\}$, and the orthogonal projection M_ϕ onto this space is such that $M_\phi h = h - \text{Re}\langle \phi, h \rangle \phi$. Writing the Euler-Lagrange equation (first-order necessary condition for the minimum) associated with our problem at a minimum $\phi \in \mathcal{S}$ requires that the projection of the gradient on the tangent space $\mathcal{S}$ is zero, i.e. $M_\phi \nabla \mathcal{E}_{\text{tot}}(\phi) = 0$. This equation is equivalent to the nonlinear eigenvalue problem

$$\mathcal{H}_\phi \phi = \lambda \phi,$$

where $\lambda := \lambda(\phi) = \langle \mathcal{H}_\phi \phi, \phi \rangle$ is the Lagrange multiplier associated to the spherical constraint (also known as the chemical potential).

2.2. Pseudospectral spatial discretization

To find a numerical solution of the minimization problem, the function $\phi \in L^2(\mathbb{R}^d)$ must be discretized carefully and accurately, most particularly to describe fine details like the vortex lattice structure that can appear. BEC2HPC considers a standard pseudo-spectral discretization scheme based on Fast Fourier Transforms (FFTs) [18,22,23,27]. In 3D, we truncate the wave function ϕ to a square domain $[-a_x, a_x] \times [-a_y, a_y] \times [-a_z, a_z]$ (a_x, a_y and a_z being positive), with periodic boundary conditions, and discretize ϕ with an even number of n_x, n_y and n_z grid points in the respective x-, y- and z-directions. We describe our scheme in 3D (the 2D case being then straightforward). For $M := (n_x, n_y, n_z)$, we introduce a uniformly sampled grid: $\mathcal{D}_M := \{\mathbf{x}_{k_1,k_2,k_3} = (x_{k_1}, y_{k_2}, z_{k_3})\}_{(k_1,k_2,k_3)\in\mathcal{O}_M}$, with $\mathcal{O}_M := \{0, \ldots, n_x - 1\} \times \{0, \ldots, n_y - 1\} \times \{0, \ldots, n_z - 1\}$, $x_{k_1+1} - x_{k_1} = h_x$, $y_{k_2+1} - y_{k_2} = h_y$ and $z_{k_3+1} - z_{k_3} = h_z$, with mesh sizes $h_x = 2a_x/n_x$, $h_y = 2a_y/n_y$ and $h_z = 2a_z/n_z$. By considering the $N \times N$ Hermitian matrix(-free) operators from $\mathbb{C}^N$ ($N = n_x n_y n_z$ in 3D) to $\mathbb{C}$ given by $[\![\Delta]\!] := [\![\partial_x^2]\!] + [\![\partial_y^2]\!] + [\![\partial_z^2]\!]$ and $[\![\mathbf{\Omega} \cdot \mathbf{L}]\!] := -i\mathbf{\Omega} \cdot (\mathbf{x} \wedge [\![\nabla]\!])$, we obtain the discretization of the gradient of the energy

$$([\![\nabla]\!] E_{\text{tot}})(\boldsymbol{\phi}) = 2[\![\mathcal{H}_{\boldsymbol{\phi}}]\!]\boldsymbol{\phi},$$

with

$$[\![\mathcal{H}_{\boldsymbol{\phi}}]\!] := -\frac{1}{2}[\![\Delta]\!] + [\![V]\!] + [\![f(|\boldsymbol{\phi}|^2)]\!] - [\![\mathbf{\Omega} \cdot \mathbf{L}]\!].$$

We set $\boldsymbol{\phi} := (\tilde{\phi}(\mathbf{x}_{k_1,k_2,k_3}))_{(k_1,k_2,k_3)\in\mathcal{O}_M}$ (where $\tilde{\phi}$ is the approximation of the function ϕ) as the discrete unknown vector in $\mathbb{C}^N$. For conciseness, we identify an array $\boldsymbol{\phi}$ in the vector space of 3D complex-valued arrays $\mathcal{M}_{n_x,n_y,n_z}(\mathbb{C})$ (storage according to the 3D grid) and the reshaped vector $\boldsymbol{\phi}$ in $\mathbb{C}^N$. In addition, to simplify the notations, we forget the brackets $[\![A]\!]$ and use A to designate the matrix operator associated with a continuous operator A, based on the FFT approximation. Finally, the cost for evaluating the application of a 3D FFT is $\mathcal{O}(N \log N)$.

For $\boldsymbol{\phi} \in \mathbb{C}^N$, the total discrete energy $E_{\text{tot}}(\boldsymbol{\phi})$ can be written as the sum of the four elementary energies

$$E_{\text{tot}}(\boldsymbol{\phi}) = E_{\text{kin}}(\boldsymbol{\phi}) + E_{\text{pot}}(\boldsymbol{\phi}) + E_{\text{int}}(\boldsymbol{\phi}) + E_{\text{rot}}(\boldsymbol{\phi}), \tag{2.4}$$

setting

$$\begin{aligned} E_{\text{kin}}(\boldsymbol{\phi}) &:= \frac{1}{2}||\nabla\boldsymbol{\phi}||_2^2 = \frac{1}{2}\langle \nabla\boldsymbol{\phi}, \nabla\boldsymbol{\phi} \rangle \\ &= \frac{1}{2}(\langle \partial_x\boldsymbol{\phi}, \partial_x\boldsymbol{\phi} \rangle + \langle \partial_y\boldsymbol{\phi}, \partial_y\boldsymbol{\phi} \rangle + \langle \partial_z\boldsymbol{\phi}, \partial_z\boldsymbol{\phi} \rangle), \\ E_{\text{pot}}(\boldsymbol{\phi}) &:= \langle V\boldsymbol{\phi}, \boldsymbol{\phi} \rangle, \quad E_{\text{int}}(\boldsymbol{\phi}) := \langle F(\boldsymbol{\rho}), \mathbf{1} \rangle, \\ E_{\text{rot}}(\boldsymbol{\phi}) &= -\text{Re}(\langle \mathbf{\Omega} \cdot \mathbf{L}\boldsymbol{\phi}, \boldsymbol{\phi} \rangle), \end{aligned} \tag{2.5}$$

where $\mathbf{1} \in \mathbb{C}^N$ is the vector with components 1, and $||\boldsymbol{\phi}||_2$ is the discretization of the $L^2(\mathbb{R}^d)$-norm on the uniform grid subject to the discrete hermitian inner product $\langle \mathbf{u}, \mathbf{v} \rangle$ for two complex-valued functions defined on the grid $\mathcal{D}_M$.

2.3. The Preconditioned Conjugate Gradient (PCG) method

Based on the pseudospectral discretization, we now need to compute the solution to the finite-dimensional minimization problem under normalization constraint

$$\boldsymbol{\phi} \in \underset{\boldsymbol{\phi}\in\mathbb{C}^N, \|\boldsymbol{\phi}\|_2=1}{\arg\min} E_{\text{tot}}(\boldsymbol{\phi}). \tag{2.6}$$

In BEC2HPC, we use the Preconditioned Conjugate Gradient (PCG) method which differs from the preconditioned gradient method by the following update rule for the descent method

$$\mathbf{d}_n = -P\mathbf{r}_n + \beta_n \mathbf{p}_{n-1}, \tag{2.7}$$

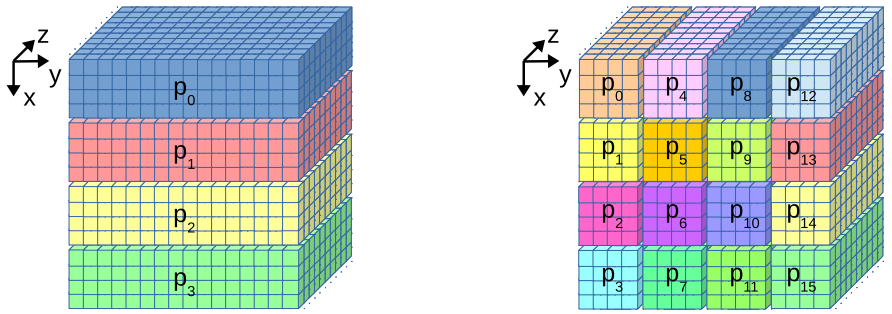
where P designates a well-designed preconditioner and the residual vector is $\mathbf{r}_n := (\mathcal{H}_{\boldsymbol{\phi}_n} - \lambda_n I)\boldsymbol{\phi}_n$. The vector $\boldsymbol{\phi}_n$ is the iterate of $\boldsymbol{\phi}$ at step n for the minimization method (a preconditioned descent algorithm would lead to the relation $\mathbf{d}_n = -P\mathbf{r}_n$). In addition, we define $\mathbf{p}_n = \mathbf{d}_n - \text{Re}\langle \mathbf{d}_n, \boldsymbol{\phi}_n \rangle \boldsymbol{\phi}_n$ as the orthogonal projection of $\mathbf{d}_n$ onto the space generated by $\boldsymbol{\phi}_n$. The step β_n is given by the Polak-Ribière formula $\beta = \max(\beta^{\text{PR}}, 0)$, setting

$$\beta^{\text{PR}} = \frac{\langle \mathbf{r}_n - \mathbf{r}_{n-1}, P\mathbf{r}_n \rangle}{\langle \mathbf{r}_{n-1}, P\mathbf{r}_{n-1} \rangle}. \tag{2.8}$$

We consider the standard choice $\beta = \max(\beta^{\text{PR}}, 0)$, corresponding to restarting by the CG method when $\beta^{\text{PR}} < 0$. For the justification of the CG method for constrained minimization, we refer to [54,55]. The CG algorithm is then summarized in Algorithm 1. Concerning the choice of θ_n, we take

$$\theta_n^{\text{opt}} = \frac{-\text{Re}\left\langle (\nabla E_{\text{tot}})(\boldsymbol{\phi}_n), \mathbf{p}_n \right\rangle \|\mathbf{p}_n\|}{\text{Re}\left[(\nabla^2 E_{\text{tot}})(\boldsymbol{\phi}_n)[\mathbf{p}_n, \mathbf{p}_n] - \lambda_n \right]}, \tag{2.9}$$

and use the same step size control as in the steepest descent algorithm. In practice, these precautions for checking the descent direction and using a stepsize control technique are important when located in the neighborhood of a minimum. When a minimum is approximately obtained, $\mathbf{p}_n$ is always a descent direction and the stepsize choice (2.9) decreases the energy functional. Finally, we use the following robust stopping criterion (see [37])



(a) A 1D domain decomposition in the X direction. (b) A 2D (pencil) domain decomposition.

Fig. 1. Decomposition strategies for a 3D domain.

while *not converged* **do**
- $\lambda_n = \lambda(\boldsymbol{\phi}_n)$
- $\mathbf{r}_n = (\mathcal{H}_{\boldsymbol{\phi}_n} - \lambda_n I)\boldsymbol{\phi}_n$
- $\beta_n = \langle \mathbf{r}_n - \mathbf{r}_{n-1}, P\mathbf{r}_n\rangle / \langle \mathbf{r}_{n-1}, P\mathbf{r}_{n-1}\rangle$
- $\beta_n = \max(\beta_n^{\mathrm{PR}}, 0)$
- $\mathbf{d}_n = -P\mathbf{r}_n + \beta_n \mathbf{p}_{n-1}$
- $\mathbf{p}_n = \mathbf{d}_n - \mathrm{Re}\langle \mathbf{d}_n, \boldsymbol{\phi}_n\rangle \boldsymbol{\phi}_n$
- $\theta_n = \arg\min_\theta E\left(\cos(\theta)\boldsymbol{\phi}_n + \sin(\theta)\mathbf{p}_n/\|\mathbf{p}_n\|\right)$
- $\boldsymbol{\phi}_{n+1} = \cos(\theta_n)\boldsymbol{\phi}_n + \sin(\theta_n)\mathbf{p}_n/\|\mathbf{p}_n\|$
- $n = n+1$

end

Algorithm 1: The Preconditioned Conjugate Gradient (PCG) method as implemented in BEC2HPC.

$$E_{\mathrm{err}}^n := |E_{\mathrm{tot}}(\boldsymbol{\phi}_{n+1}) - E_{\mathrm{tot}}(\boldsymbol{\phi}_n)| \leq \varepsilon. \tag{2.10}$$

This finally leads to the following PCG algorithm.

Let us now focus on the important question of building an efficient and robust preconditioner P as a modification of the descent direction, leading then to a closer point to the minimum

$$\mathbf{d}_n := -P(\mathcal{H}_{\boldsymbol{\phi}_n} - \lambda_n I)\boldsymbol{\phi}_n. \tag{2.11}$$

The preconditioner should be an approximation of the inverse of the Hessian matrix of the problem. A first preconditioner [37] uses only the kinetic energy term through an adaptive preconditioner

$$P_\Delta^n = (\alpha_\Delta^n I - \Delta/2)^{-1}, \tag{2.12}$$

where α_Δ^n is a positive shifting constant defined by the characteristic energy

$$\alpha_\Delta^n = E_{\mathrm{kin}}(\boldsymbol{\phi}_n) + E_{\mathrm{pot}}(\boldsymbol{\phi}_n) + E_{\mathrm{int}}(\boldsymbol{\phi}_n) > 0. \tag{2.13}$$

This preconditioner provides a convergence independent of the grid refinement and is diagonal in the Fourier space. Another natural approach is to use the diagonal preconditioner based on the potential and nonlinear interaction terms

$$P_V^n = (\alpha_V^n I + V + f(|\boldsymbol{\phi}_n|^2))^{-1}, \tag{2.14}$$

with $\alpha_V^n = \alpha_\Delta^n$. This preconditioner is well-adapted to large nonlinearities and large domains. Finally, to get a stable performance independent of the size of the domain or the spatial resolution, we can define the combined preconditioners

$$P_{C_1}^n = P_V^n P_\Delta^n, \quad P_{C_2}^n = P_\Delta^n P_V^n \tag{2.15}$$

or a symmetrized version

$$P_C^n = P_V^{n,1/2} P_\Delta^n P_V^{n,1/2}. \tag{2.16}$$

Let us now analyze the computational cost of the PCG. The application of the operator P_V is almost free (since it only requires a scaling of $\boldsymbol{\phi}$), but the naive application of P_Δ requires a FFT/IFFT pair. However, since we apply the preconditioners after and before an application of the Hamiltonian, we can reuse the FFT and IFFT computations, so that the application of P_Δ does not require any additional Fourier transform. Similarly, the use of P_{C_1} and P_{C_2} only needs one additional Fourier transform per iteration, and that of the symmetrized version P_C two.

To summarize, the cost in terms of Fourier transforms per iteration for the rotating GPE model is

- no preconditioner: 3 FFTs/iteration (get the Fourier transform of $\boldsymbol{\phi}$, and two IFFTs to compute $\Delta\boldsymbol{\phi}$ and $L_z\boldsymbol{\phi}$ respectively),
- P_Δ or P_V: 3 FFTs/iteration,
- non-symmetric combined P_{C_1} or P_{C_2}: 4 FFTs/iteration,
- symmetric combined P_C: 5 FFTs/iteration.

Note that this total cost might be different for another type of GPE model e.g. when a nonlocal dipole-dipole interaction is included [18,53].

3. Implementation of the PCG method

In this section, we present an implementation of the previous PCG method for computing the stationary states of the GPE. The resulting code, called BEC2HPC, is developed in C++ and uses MPI communications for distributed computing. The code is available online at https://team.inria.fr/bec2hpc/. A Python interface is provided for defining the physics of the problem and external visualization tools such as Paraview can be used to exploit the results of the simulations. The code takes advantage of existing HPC libraries and even if we focus here on the implementation of the PCG methods, it can itself be leveraged to implement other spectral methods or solve e.g. problems related to the dynamics of BECs. In particular, schemes for simulating the dynamics of the GPE will be included in a future version of BEC2HPC.

3.1. Implementation of FFT-based schemes in distributed memory

The wave function ϕ is truncated to a square domain with periodic boundary conditions and discretized on a grid with n_x, n_y, n_z points along each dimension x, y and z, respectively. This grid structure leads to multidimensional arrays on which we perform one-dimensional discrete Fourier transforms along each dimension independently. For a 2D domain, we compute the transform of each column and each row of a matrix to obtain $[[\partial_x]]$ and $[[\partial_y]]$. In distributed-memory, these arrays are divided among a set of processes which each runs in their own memory address space. Several libraries implement parallel FFT algorithms working on distributed data. These codes use either 1D slab (FFTW [56]) or 2D pencil decompositions (PFFT [57], P3DFFT [58], 2DECOMP&FFT [59]). The idea is to simply reuse serial FFT algorithms in local memory. A 1D domain distribution along the dimension X (Fig. 1)

of a 3D domain allows to compute both $[[\partial_y]]$ and $[[\partial_z]]$ without any interprocess communications as the grid elements involved in the computation of each 1D FFT along the Y and Z lines are assigned to the same process. A permutation of the axis of the array makes the computation along X local at the cost of all-to-all communication to transpose the distributed array. A 2D pencil domain decomposition only maintains the data placement useful for one direction and requires transpositions for each of the other directions but it exhibits more parallelism as a 1D decomposition along the dimension X is limited to using at most n_x processes.

BEC2HPC is based on FFTW 3.3 (see [56]). We use the 1D block distribution of the data (distributed along the first dimension), the sequential FFT routine and the distributed transpose routine provided by the library. Using the FFTW's advanced interface, one can perform multiple sequential 1D complex FFT simultaneously on non-contiguous data (`fftw_plan_many_dft`), allowing to compute directly the FFTs along Y and Z on each process. The MPI transpose routine used internally by FFTW for multidimensional transforms is also exposed in its API (`fftw_mpi_plan_many_transpose` for complex numbers) and is called directly for the computation of $[[\partial_x]]$. FFTW comes with a set of routines for the creation of the domain distribution (dividing the data among the MPI processes), the allocation of memory (with due consideration to alignment for SIMD instructions and the extra memory that might be needed for a data redistribution). Computation of a Fourier transform is preceded by a preprocessing phase selecting at runtime an efficient strategy for computing a transform on the current hardware: it creates a *plan* (an opaque data type) describing the algorithm and transforms can then be executed repeatedly.

3.2. BEC2HPC parallel code design

Building FFT-based schemes on the foundation of FFTW is a good starting point as this library is fast, freely available and became defacto standard for scientific softwares or for benchmarking other FFT libraries. In addition, other FFT libraries such as the Intel Math Kernel Library (MKL) offers FFTW interfaces without changing the program source code. NVIDIA also provides FFTW interfaces to the cuFFT library. However, it seems important to integrate in the initial design more general data distributions as well as the possibilities of using hybrid MPI-plus-thread approach in the future. On multicore clusters, OpenMP directives can be used to distribute the set of serial 1D FFT to be performed within a MPI process. Multi-threaded FFT algorithms can also replace serial FFT to leverage parallelism along a second dimension as FFTW provides multi-threaded transforms with exactly the same API as the serial version.

Currently, BEC2HPC only uses FFTW but we avoided binding the code directly to this third-party library. FFTW's routine calls and data types are encapsulated and BEC2HPC defines its own interface for data distribution, transforms and transposition. In BEC2HPC, the data distribution is described by a `Map` object taking up ideas of well established parallel scientific libraries [60]. `Map` objects contain the details of the block distribution along each dimension (including domain dimensions, local and global partitioning indices) and distributed objects are subsequently created from `Map` objects. The distributed arrays class gives methods for creating multidimensional arrays, accessing local array elements. It also provides *foreach loops* for performing pointwise operations for each domain element and parallel reduce using C++11 lambda. It hides much of the complexity of the data distribution and facilitates the application of operators and the computation of norms and convergence criteria. As a result, the grid distribution, the local indices and MPI routine do not appear on the implementation of the PCG algorithm, improving the code readability. These classes also provide a high-level parallel abstraction layer that can evolve to support different memory layout and programming paradigm for multicore processors.

The spectral numerical method itself is implemented on top of this using a modular design to ease the change of components such as preconditioners or stopping criterias as described in Section 2. At each iteration, transforms are never computed twice and operators such as $[[\Delta]]$ or $[[L_z]]$ are stored along side ϕ_n to be used on each components.

4. Getting started with BEC2HPC

This section proposes a first BEC2HPC example and shows off the basics usage of BEC2HPC. BEC2HPC provides both a C++ and python interfaces. In this paper, we present the Python interface built using the Boost Python Library, a framework for exposing the C++ classes functions and objects to Python.

4.1. Installation

The build process of BEC2HPC is managed by CMake and requires a C++11 compiler, a MPI library and several external libraries compiled with parallel support (namely Boost, FFTW and HDF5). Python3 is also needed for running most of the examples. These softwares are usually pre-installed on HPC machines. We also provide a Vagrant setup to automatically configure a virtual computing environment suitable for compiling and running BEC2HPC on personal computers. Vagrant [61] is a popular tool for building virtual machines (or containers) from a configuration file describing the machine setup and the necessary steps to create a ready-to-use machine. The virtual machine behaves like a separate computer system and can be accessed via a SSH connection as if it was a remote physical machine. Fig. 2 shows how to create a Vagrant managed virtual machine for BEC2HPC. Within the virtual machine, the `bec2hpc` directory is shared with the host and therefore contains the source code. `build` is an out-of-source build directory to keep separate the files generated by the compilation.

4.2. A first example

Let us consider the physical problem governed by the following GPE

$$i\partial_t\psi(\mathbf{x},t) = (-\frac{1}{2}\Delta + V(\mathbf{x}) + \beta|\psi(\mathbf{x},t)|^2 - \mathbf{\Omega}\cdot\mathbf{L})\psi(\mathbf{x},t), \quad (\mathbf{x},t)\in\mathbb{R}^d\times\mathbb{R}^{*+}, \tag{4.17}$$

where $V(\mathbf{x})$ is the external confining potential, β is the nonlinearity strength describing the interaction between atoms of the condensate, $\mathbf{\Omega}$ is the angular velocity vector and $\mathbf{L}$ is the angular momentum operator. By default in BEC2HPC, the rotation term is such that $\mathbf{\Omega}\cdot\mathbf{L} = \omega L_z = -i\omega(x\partial_y - y\partial_x)$ (i.e. $\mathbf{\Omega} = (0,0,\omega)^t$), and the nonlinearity is cubic, i.e. $f(|\psi|) = \beta|\psi|^2$. The harmonic potential V is given by

$$V(x,y,z) = \frac{1}{2}(\gamma_x^2x^2 + \gamma_y^2y^2 + \gamma_z^2z^2), \tag{4.18}$$

with $\gamma_x = \gamma_y = \gamma_z = 1$ per default in 3D, and $\gamma_x = \gamma_y = 1, \gamma_z = 0$ in 2D. The predefined initial guess is the Thomas-Fermi approximation

$$\phi_0 = \frac{\phi_\beta^{\mathrm{TG}}}{||\phi_\beta^{\mathrm{TG}}||_2}, \text{ with } \phi_\beta^{\mathrm{TG}} = \begin{cases} \sqrt{\dfrac{\mu_\beta^{\mathrm{TG}} - V(\mathbf{x})}{\beta}}, & \text{if } \mu_\beta^{\mathrm{TG}} > V(\mathbf{x}), \\ 0, & \text{otherwise,} \end{cases} \tag{4.19}$$

```
# install vagrant and virtualbox
user@localhost:~/$ sudo apt-get install vagrant virtualbox

# build the vagrant machine
user@localhost:~/bec2hpc$ cd install             # location of the Vagrantfile
user@localhost:~/bec2hpc/install$ vagrant up     # create and configure the VM
user@localhost:~/bec2hpc/install$ vagrant -Y ssh # log into the VM

# compile bec2hpc within the VM
vagrant@bec2hpc-machine:~$ mkdir build; cd build; cmake ../bec2hpc/; make

# stop and delete the VM
vagrant@bec2hpc-machine:~$ exit                  # exit the SSH session
user@localhost:~/bec2hpc/install$ vagrant halt   # stop the VM
user@localhost:~/bec2hpc/install$ vagrant delete # delete the VM
```

Fig. 2. Creating a Vagrant managed Virtual Machine (VM) for BEC2HPC.

```
#!/usr/bin/env python

import mpi4py.MPI as mpi
import bec2hpc as solver

comm = mpi.COMM_WORLD

physics_params = {'Lx': 16, 'Ly': 16,    # Computational domain [-L, L]
                  'nx': 128, 'ny': 128, # Number of grid points
                  'omega': 0.5,          # Rotation speed
                  'beta': 500}           # Nonlinearity strength

solver_params  = {'verbose': True, 'stopping_criteria': 1e-12}

solver_phi_0 = solver.physics.initial_guess(comm, physics_params)

solver_phi_n, stats = solver.pcg(solver_phi_0, physics_params, solver_params);
if comm.rank == 0: print stats
```

Fig. 3. A first example.

where the eigenvalue approximation μ_β^{TG} is given by

$$\mu_\beta^{\mathrm{TG}} = \frac{1}{2}\begin{cases} (4\beta\gamma_x\gamma_y/\pi)^{1/2}, & d=2, \\ (\dfrac{15}{4\pi}\beta\gamma_x\gamma_y\gamma_z)^{2/5}, & d=3. \end{cases}$$

The stopping criterion is fixed by default to $\mathcal{E}_{\mathrm{err}}^n := |E_{\mathrm{tot}}(\phi_{n+1}) - E_{\mathrm{tot}}(\phi_n)| \le \varepsilon$, with $\varepsilon = 10^{-12}$.

We consider now that we want to compute the ground state of a 2D rotating condensate with a cubic nonlinearity and a harmonic potential (setting $\gamma_x = \gamma_y = 1$, $\gamma_z = 0$). Fig. 3 shows a first example of the BEC2HPC API. The computational domain and spatial mesh sizes are chosen respectively as $[-16, 16]^2$ and $h = \frac{1}{4}$ ($M = 128$). This example can be run within the virtual machine with `mpiexec -n 1 python simple-example.py`. The physics and solver parameters are listed in two distinct python hashmaps. The `physics` submodule can be used to generate the initial guess. The `pcg` method returns the solution `solver_phi_n` along with information about the solver execution in a hashmap `stats`. This includes in particular the energy at the final state, the convergence history and the computational time to reach the stationary state (Fig. 4). The solution `solver_phi_n` can be saved in a HDF5 file as shown in Fig. 5. HDF5 [62] is a widely-used standard binary format for storing numerical data and BEC2HPC uses the Parallel HDF5 library to efficiently write on disk in a parallel environment (using MPI-I/O). The solution can later be postprocessed or visualized with a variety of tools such as Python or Paraview (see also Section 5.3). The solution can also be transferred to a single processor by using `local_phi = solver.array.gather(phi)`. For instance, it can be used to plot `phi_n` without saving it to a file. Fig. 6 shows how to load a HDF5 file and plot the density function $\rho = |\phi|^2$ in Python. The provided function `bec2hpc.utils.plot2d` uses matplotlib internally. Fig. 7 shows the square of the amplitude of the wave function on the computational domain for $\omega = 0.5$ and $\beta = 500$. The `stats` output of `solver.pcg` can be saved as a JSON file (or in any human-readable format) along side the input parameters to retain the information of a simulation run.

5. BEC2HPC advanced usage

After this step-by-step example on a model problem, we now describe in more details some of the code functionalities for defining the physical problem and the numerical scheme. We also present along the way how to manage the distributed data structures efficiently in Python.

5.1. Defining the initial data

The iterative method for computing the solution of the minimization problem needs to be initialized with a guess. As the minimization algorithm is a local optimization procedure, the choice of the initial guess can lead to a local minimum and therefore a

```
{
    "iteration_count": 1087,
    "mass": 0.9999999999999961,
    "energy": 8.024610125450689,

    "energy_truncate_err": 8.775202786637237e-13,

    "solve_time": 162.915664,

    "energies": {
        "total_energy": 8.024610125450689,
        "kinetic_energy" 1.2959104572267384,
        "potential_energy": 4.97235188442656,
        "interaction_energy": 3.6767245784205898,
        "rotation_energy": -1.9203767946231975
        "chemical_potential": 11.701334703871279,
    }

   "iterations":[
       {
         "mass": 1.0000000000000004,
         "energy": 8.6049880789089634,
         ...
       },
       { ... },
   ]
}
```


Fig. 4. Output of the `solver.pcg` function in a JSON format.

```
import json

solver.utils.save_hdf5(solver_phi_n, 'phi_n.h5')

if comm.rank == 0:
    with open('run.json', 'w') as fd:
        json.dump(
            {'physics_params': physics_params,
             'solver_params': solver_params,
             'solver_stats': stats},
            fd,
            sort_keys=True, indent=2)
```

Fig. 5. Saving the ouput of the `solver.pcg` function.

different final converged state [37]. The initial guess is usually an approximation of the solution of a simpler problem and BEC2HPC provides initial data typically found in the literature such as a centered Gaussian for fast rotations or the Thomas-Fermi approximations (4.19) for strong nonlinear interactions [18,36,37]. In 2D, the centered Gaussian is defined by

$$\phi(\mathbf{x}) = \frac{(1-\omega)\phi_a(\mathbf{x}) + \omega\phi_b(\mathbf{x})}{||(1-\omega)\phi_a(\mathbf{x}) + \omega\phi_b(\mathbf{x})||}, \tag{5.20}$$

where

$$\phi_a(\mathbf{x}) = \frac{1}{\sqrt{\pi}} e^{-(\gamma_x x^2 + \gamma_y y^2)/2}, \qquad \phi_b(x) = (\gamma_x x + i\gamma_y y)\phi_a(\mathbf{x}). \tag{5.21}$$

It is also possible to provide your own initial guess to the `pcg` method once you become familiar with the BEC2HPC distributed array class. Fig. 8 shows how to implement the Gaussian defined

```
#!/usr/bin/env python

import numpy
import h5py
import bec2hpc

# Load HDF5 file
file = h5py.File('phi_n.h5', 'r+')
dataset = file['/phi']
a = numpy.array(dataset)
phi = a.view(dtype=numpy.complex128)

# Compute |phi|^2
phi = numpy.absolute(phi)**2

# Plot |phi|^2 (using matplotlib)
bec2hpc.utils.plot2d(phi)
```

Fig. 6. Compute and plot the density function $|\phi|^2$.

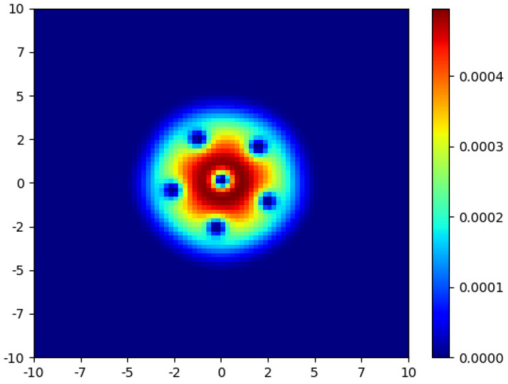


Fig. 7. Density function $|\phi|^2$ of the ground state.

by (5.20) for $\gamma_x = \gamma_y = 1$ and $\omega = 0$, a simple choice for weak nonlinear interaction and subcritical frequencies.

In Python, the elements of a `DistributedArray` are accessible as a numpy array by using the `getData()` method. This method returns a numpy array containing the local part of the distributed array without making a copy of the underlying data. The numpy array is actually a *view* of the original data and modifying the numpy array changes the distributed array (and vice versa). The data distribution of an array is described by a `Map` object and can be retrieved using the `getMap()` method on the array. The python ellipsis syntax ('...') is used to assign the result of the calculation to the original numpy array memory buffer (otherwise, with a simple array assignment, `phi_0` would not be referencing the BEC2HPC array anymore).

5.2. Defining the trapping potential

BEC2HPC comes with predefined potential functions for an harmonic trap (4.18) and other potentials with added potential terms such as the harmonic-plus-quartic potential

$$V(\mathbf{x}) = (1-\alpha)\sum_{\nu=x,y} \gamma_\nu \nu^2 + \frac{\kappa\,(x^2+y^2)^2}{4} \begin{cases} +0, & d=2, \\ +\gamma_z^2 z^2, & d=3. \end{cases} \tag{5.22}$$

Fig. 9 shows how to provide a user-defined potential in Python. Either Python function or a lambda form could be specified on

```python
#!/usr/bin/env python

import numpy
import mpi4py.MPI as mpi
import bec2hpc as solver

def initial_guess(comm, nx, ny, Lx, Ly):
  map = solver.Map(comm, nx, ny, Lx, Ly)
  solver_phi_0 = solver.DistributedArray(map)

  phi_0 = solver_phi_0.getData()

  x = -Lx + numpy.arange(map.local_nx_start(),
                         map.local_nx_start() + map.local_nx()) * map.hx()
  y = -Ly + numpy.arange(map.local_ny_start(),
                         map.local_ny_start() + map.local_ny()) * map.hy()
  X, Y = numpy.meshgrid(x, y)

  # [...] assign the array in-place
  phi_0[...] = numpy.exp(-(X**2 + Y**2) / 2) / numpy.sqrt(numpy.pi)

  # normalization
  l_mass = numpy.sum(numpy.absolute(phi_0)**2)
  mass = comm.allreduce(l_mass,op=mpi.SUM)
  phi_0[...] = phi_0 / numpy.sqrt(mass * map.hx() * map.hy())

  return solver_phi_0
```

Fig. 8. Defining a initial guess.

```python
# Using a predefined potential (written in C++)
physics_params = {'Lx': Lx, 'Ly': Ly, 'nx': nx, 'ny': ny,
                  'potential': solver.physics.quadratic_potential(gamma_x = 1,
                                                                  gamma_y = 1)}

# Using a user-defined potential (python function)
def my_potential(x, y):
    return (x**2 + y**2)/2;

physics_params = {'Lx': Lx, 'Ly': Ly,
                  'nx': nx, 'ny': ny,
                  'potential': my_potential}

# Using a user-defined potential (lambda function)
physics_params = {'Lx': Lx, 'Ly': Ly,
                  'nx': nx, 'ny': ny,
                  'potential': lambda x, y: (x**2 + y**2)/2}
```

Fig. 9. User-specified potentials.

the parameter list describing the physics of the problem. One can also implement its own C++ potential class by deriving the C++ `Potential` abstract class and instantiating it in Python. On the same model, it is possible to redefine the nonlinearity part of the equation as well as the stopping criteria and the preconditioner to be used during the nonlinear conjugate gradient.

5.3. 3D simulation and visualization

The previous examples were in 2D for conciseness but BEC2HPC has been designed with large 3D problems in mind. For code maintenance purpose, 2D and 3D cases share the same code base internally (there is no code duplication). The functions described in 2D in this paper are all available in 3D as well. Running a 3D simulation only requires to define the computational boundary and the number of grid points in the third dimension (by adding `Lz` and `nz` in the `physics` parameter list).

3D visualizations of large problems might be more tricky but thankfully, specialized applications such as ParaView can be leveraged [63]. ParaView cannot read directly a HDF5 file as it needs information concerning the semantic of the data. Such information can be provided by a simple XDMF file describing the data scheme. XDMF (eXtensible Data Model and Format) uses XML to store metadata and refers to external HDF5 files for the values themselves. It is a standard way format to describe the raw data produced by HPC codes and BEC2HPC provides a function `bec2hpc.utils.save_xdmf(phi_n, 'phi_n.xdmf')` to create a XDMF describing the semantic of a BEC2HPC HDF5 output

created with the call `bec2hpc.utils.save_hdf5(phi_n, 'phi_n.hdf5)`.

The ParaView Python API gives full access to its data analysis and visualization capabilities. It can be used to programmatically visualize the result of a BEC2HPC simulation. When running multiple tests, it might be handy to define a visualization with the ParaView application, save the visualization state and then reload it from Python. An example to load a ParaView state file (a file with a `.pvsm` extension) is available in the `examples/paraview` directory of BEC2HPC.

5.4. Grid manipulation

In order to both limit the number of iterations and reach the lowest energy state, the initial guess of a simulation must be chosen close to the expected solution. Several choices for an initial guess are described in subsection 5.1. Another approach to define an initial guess is to use the results of a previous simulation. It can give good results when the simulation parameters varies only slightly or when we make them gradually changing. For large grid sizes, one can use the results of the same simulation performed on coarser grid involving less points. It is also possible to tune the size of the domains once we learned on a fastest, less accurate simulation the space occupied by the condensate for a given rotational speed and potential.

The mapping of a coarse grid data on a finer grid is done using an interpolation. Fig. 10 gives an example for linearly interpolating a 2D grid in Python. In this example, the coarse grid solution is simply gathered on a single processor and the interpolation is done in sequential using the `interpolate.interpn` function of the SciPy library. This process can be repeated on several grid levels to form the hierarchical multigrid strategy presented in [36,37] and implemented with BEC2HPC in Fig. 11. The ground state computation begins on a coarse grid of $2^{N_{\text{coarse}}} \times 2^{N_{\text{coarse}}}$ points and the grid is successively refined until a finest grid with $2^{N_{\text{fine}}} \times 2^{N_{\text{fine}}}$. The stopping criteria can be relaxed for the coarse grids as shown in the example.

Nevertheless, strategies based on simple continuation or multigrid strategies should be used carefully since they can also sometimes provide intermediate or final solutions which are not correct. More advanced algorithms [20,30,64] should probably be considered to obtain fast and robust methods.

6. Numerical examples

6.1. Experimental setup

In this section, we present some results obtained from simulations with BEC2HPC in 2D/3D for fast rotating BECs on a parallel cluster. Let us remark that the aim of the paper is not to compare BEC2HPC to other solvers. Nevertheless, it would be interesting to have some fair comparisons between codes, and in particular with [47,51] where efficient solutions for simulating rotating BECs are given. Let us also recall that, from [37], it is already stated that the PCG is much more efficient than an optimized normalized gradient flow/imaginary time method with pseudospectral approximation. Experiments were conducted on NIC4, a massively parallel cluster of the University of Liège, installed in the framework of the Belgium consortium of HPC centers (**CÉCI**) and funded by F.R.S.-FNRS under Grant No. 2.5020.11. It features nodes with two 8-cores Intel E5-2650 processors at 2.0 GHz and 64 GB of RAM (4 GB/core), interconnected with a QDR Infiniband network. We focus here on the strong scaling (how the solution time varies with the number of processors for a fixed total problem size) and reach indeed a limit where each MPI processes does not have much computing work to do. The scalability problem that occurs with very small problem sizes per processor by using hybrid MPI/OpenMP optimization will be addressed in the future.

In the following examples, we consider the cubic nonlinearity and the harmonic-plus-quartic potential (5.22), with $\gamma_x = \gamma_y = 1$, $\alpha = 1.2$ and $\kappa = 0.3$. In 3D, we fix $\gamma_z = 3$. This potential leads to the existence of stationary states for highly rotating BECs for values ω larger than one, unlike the standard harmonic potential case. We initialize the PCG method with the Thomas-Fermi ansatz (4.19) presented in Section 4.2. We only report the results obtained with the symmetrical version of the combined preconditioner (2.16) as it outperforms the other tested preconditioners. For the stopping criterion, we use $\mathcal{E}^n_{\text{err}} := |E(\phi_{n+1}) - E(\phi_n)| \leq \varepsilon$ which is well suited for rotating BECs (to include the non-uniqueness of the minimum up to a rotation). Let us note here that all the computations presented here were already checked in [37] and therefore represent some stationary states, which are expected to be some the ground state of the problem under study. However, the PCG remains a local minimization algorithm (similarly to imaginary time methods) and therefore the iterated solution may converge to a local minimizer which is a not the ground state (see also [37] for explicit examples). This is a fundamental limit of the local optimization techniques under constraint. In the future, improvements of these local methods should be developed to have more robust solutions for the most complex cases. In particular, let us mention that two initial guess may lead to two different solutions with very close total energies.

The experiments presented here were carried out for a wide range of rotation speeds and nonlinearity strengths. The strong scaling of the code is also tested by running the same experiments with an increasing number of MPI processes. To facilitate the testing when several parameters need to vary and to run the solver several times, we provide an easy way to generate and launch a set of experiments. These scripts are included in the `example` directory of the BEC2HPC distribution. Fig. 12 shows how to generate input parameters lists describing the physics for each combination of ω and β where $\omega \in \{1, ..., 4.5\}$ and $\beta \in \{1000, 5000, 10000\}$. The Python dictionary `physics_params_set` includes a description of the parameters range for ω and β. `utils.extend_params` generates all possible combinations of the given input parameters for the `pcg` function. In this example, each parameters list is saved in its own JSON file and the utility function `gen_id` generates a string to name the experiments (for example, `2D__L_20__n_640__Omega_1__Beta_1000.json`). These JSON files can then be used to run BEC2HPC as shown in Fig. 13. This last script takes a list of JSON files as input parameter and can be used within a submission script for a job scheduler on a HPC machine.

6.2. Numerical results in 2D

In 2D, the computational domain and mesh sizes are chosen respectively as $[-20, 20]^2$ and $h = \frac{1}{16}$ ($M = 640$). Concerning the stopping criterion, the tolerance is set to $\varepsilon = 10^{-14}$. Fig. 14 shows the converged stationary states computed by BEC2HPC for $\beta = 1000$ and different rotation speeds. We also provide the value of the chemical potential λ for each case. By increasing the rotation velocity, the Bose-Einstein condensate expands into a ring shaped BEC with an increasing central radius. In particular, for $\omega = 5$, we get a thin ring with one layer of uniformly spaced vortices. Fig. 15 shows additional results for a larger nonlinearity strength, i.e. $\beta = 10000$, showing that more vortices characterize the BEC for larger values of β.

In Table 1, the number of MPI processes is fixed to $n = 32$. We compare the CPU times (in seconds) needed to have the PCG method converging, for various values of ω and β. The CPU time is directly related to the number of iterations #it. In particular,

```
#!/usr/bin/env python

import numpy as np
from scipy import interpolate
import mpi4py.MPI as mpi
import bec2hpc as solver

# 2D grid coordinates based on a map
def grid(map):
  x = -map.Lx() + np.arange(map.local_nx_start(),
                            map.local_nx_start() + map.local_nx()) * map.hx()
  y = -map.Ly() + np.arange(map.local_ny_start(),
                            map.local_ny_start() + map.local_ny()) * map.hy()
  return x, y

# 2D coarse grid interpolation
def grid_interpolate(coarse_solver_phi_par, fine_solver_phi_par):
  # Create the sequential arrays used for the interpolation on proc 0
  if comm.rank == 0:
    coarse_map = solver.Map(mpi.COMM_SELF, coarse_solver_phi_par->map())
    coarse_solver_phi = solver.DistributedArray(coarse_map)
    fine_map = solver.Map(mpi.COMM_SELF, fine_solver_phi_par->map())
    fine_solver_phi = solver.DistributedArray(fine_map)
  else:
    coarse_solver_phi = fine_solver_phi = None

  # Gather coarse grid data to proc 0
  solver.utils.gather(coarse_solver_phi_par, coarse_solver_phi)

  if comm.rank == 0:
    # Add the boundary points to the coarse grid in preparation for the interpolation
    x, y = grid(coarse_map)
    x = np.concatenate((x, [coarse_map.Lx()]), axis=0)
    y = np.concatenate((y, [coarse_map.Ly()]), axis=0)
    coarse_grid = (x, y)
    # Expand to the boundary value at Lx, Ly
    coarse_phi = np.copy(coarse_solver_phi.getData())
    coarse_phi = np.concatenate((coarse_phi, [coarse_phi[0,:]]), axis=0)
    coarse_phi = np.concatenate((coarse_phi, np.array([coarse_phi[:,0]]).T), axis=1)
    # Build the fine grid
    x1, y1 = grid(fine_map)
    X1, Y1 = np.meshgrid(x1, y1)
    fine_grid = np.array([X1, Y1]).T
    # Linear interpolation
    fine_phi = fine_solver_phi.getData()
    fine_phi[...] = interpolate.interpn(coarse_grid, coarse_phi.real, dest,
                                        bounds_error=True, fill_value=None) + \
                    1j * interpolate.interpn(coarse_grid, coarse_phi.imag, dest,
                                             bounds_error=True, fill_value=None)
    # Normalization
    l_mass = np.sum(np.absolute(fine_phi)**2);
    mass = l_mass
    mass = comm.allreduce(l_mass,op=mpi.SUM)
    fine_phi[...] = fine_phi / np.sqrt(mass * fine_map.hx() * fine_map.hy())

  # Scatter the fine grid data
  solver.scatterDistributedArray(fine_solver_phi, fine_solver_phi_par)
```

Fig. 10. Coarse grid interpolation.

higher rotation speeds ω and stronger nonlinearities β usually require more iterations (and so more simulation times) but it is not always the case. Using a stopping criteria based on the energy difference is advantageous to avoid situations where the residual evolves without changing the energy.

Table 2 presents the number of iterations #it and the CPU time required to compute the ground state for $\beta = 1000$ and different numbers of MPI processes. This shows how the computational time scales with the number of MPI processes. Running the computation on 32 cores is 15-20 times faster than on a single core. The

```
if __name__ == '__main__':
  comm = mpi.COMM_WORLD

  NCoarse = 5 # 2^5 = 32, coarest mesh size
  NFine = 9   # 2^9 = 512, finest mesh size

  physics_params_template = {'Lx': 16, 'Ly': 16,
                             'omega': 0.5,
                             'beta': 500}

  solver_params        = {'verbose': True, 'stopping_criteria': 1e-10}
  finest_solver_params = {'verbose': True, 'stopping_criteria': 1e-14}

  for N in range(NCoarse, NFine+1):
    physics_params = physics_params_template.copy()
    physics_params['nx'] = pow(2, N)
    physics_params['ny'] = pow(2, N)

    if N == NFine:
      solver_params = finest_solver_params

    # Define phi_0
    if N == NCoarse:
      # phi_0 on the coarsest grid
      solver_phi_0 = solver.physics.initial_guess(comm, physics_params)
    else:
      # phi_0 is an interpolation of the previous phi_n
      solver_phi_0_map = solver.Map(comm,
                                    physics_params['nx'], physics_params['ny'],
                                    physics_params['Lx'], physics_params['Ly'])
      solver_phi_0 = solver.DistributedArray(solver_phi_0)
      grid_interpolate(solver_phi_n, new_solver_phi_0)

    # Compute the ground state on the current grid
    solver_phi_n, solver_stats = solver.pcg(solver_phi_0,
                                            physics_params, solver_params)
```

Fig. 11. A multigrid approach for defining the initial guess.

```
#!/usr/bin/env python

import bec2hpc as solver
from bec2hpc import utils

physics_params_set = {
    'Lx' : 20, 'Ly' : 20,
    'nx' : 640, 'ny' : 640,
    'omega': utils.seq(1, 0.5, 4.5), 'beta': [1000, 5000, 10000]
}

physics_params_set = utils.extend_params(physics_params_set)

for physics_params in physics_params_set:
    run_id = utils.gen_id(physics_params)
    utils.save(physics_params, run_id + '__params.json')
```

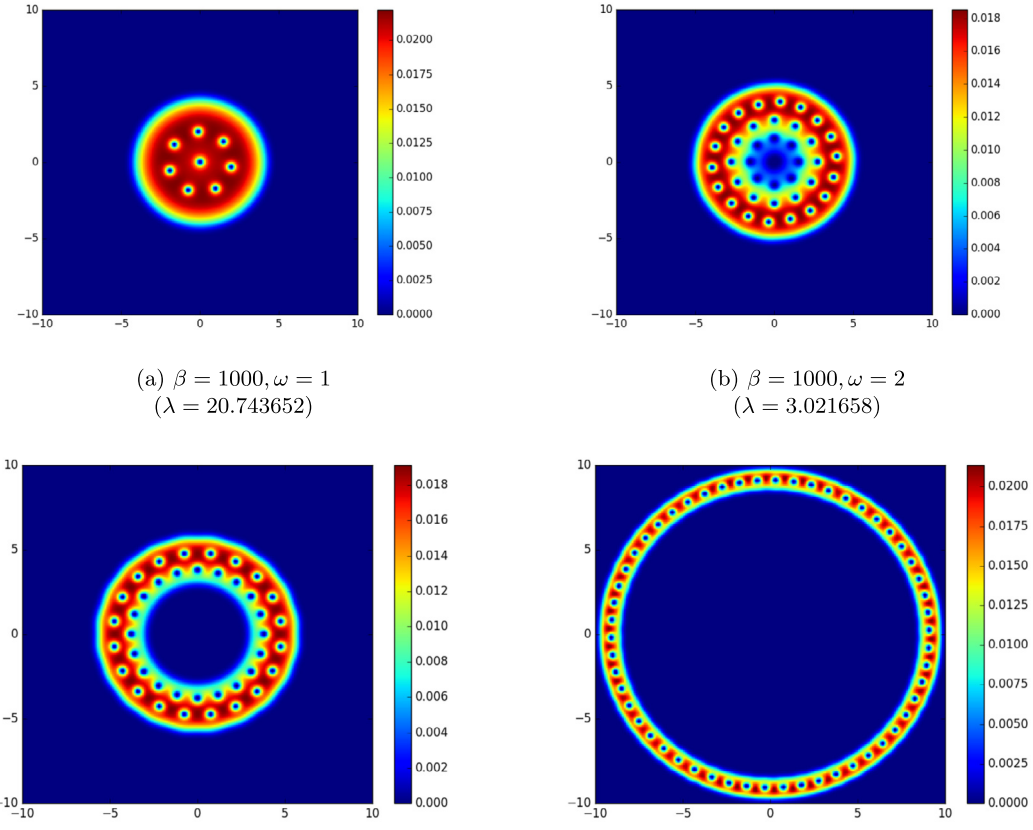
Fig. 12. Generating a JSON file for each combination of the parameters ω and β.

scalability results are consistent with the scalability of the 2D FFT algorithm in distributed memory. The number of iterations varies slightly with the number of processors but the same final state is nonetheless reached independently of the number of MPI processes.

Finally, Table 3 shows the resolution of a very difficult 2D problem ($\beta = 10000$, $\omega = 5$) on a finer grid ($M = 2048$) using 32 and 128 processors. A fast and precise resolution of such challenging problems is only possible thanks to the robustness of the numerical method and its parallel implementation.

```
import mpi4py.MPI as mpi
import bec2hpc as solver
from bec2hpc import utils

comm = mpi.COMM_WORLD

for i in range(1, len(sys.argv)):
    physics_params = utils.load(sys.argv[i])
    solver_params  = {'verbose': True, 'stopping_criteria': 1e-12}

    run_id = utils.gen_id(physics_params) + '__proc_' + str(comm.size)
    solver_phi_0 = solver.physics.initial_guess(comm, physics_params)
    solver_phi_n, solver_stats = solver.pcg(solver_phi_0,
                                            physics_params, solver_params)

    if comm.rank == 0:
        utils.save(solver_stats, run_id + '__stats.json')
    solver.save_hdf5(solver_phi_n, run_id + '__phi_n.h5')
```

Fig. 13. Running BEC2HPC using JSON files as the description of the physical setups.



(a) $\beta = 1000, \omega = 1$ ($\lambda = 20.743652$)

(b) $\beta = 1000, \omega = 2$ ($\lambda = 3.021658$)

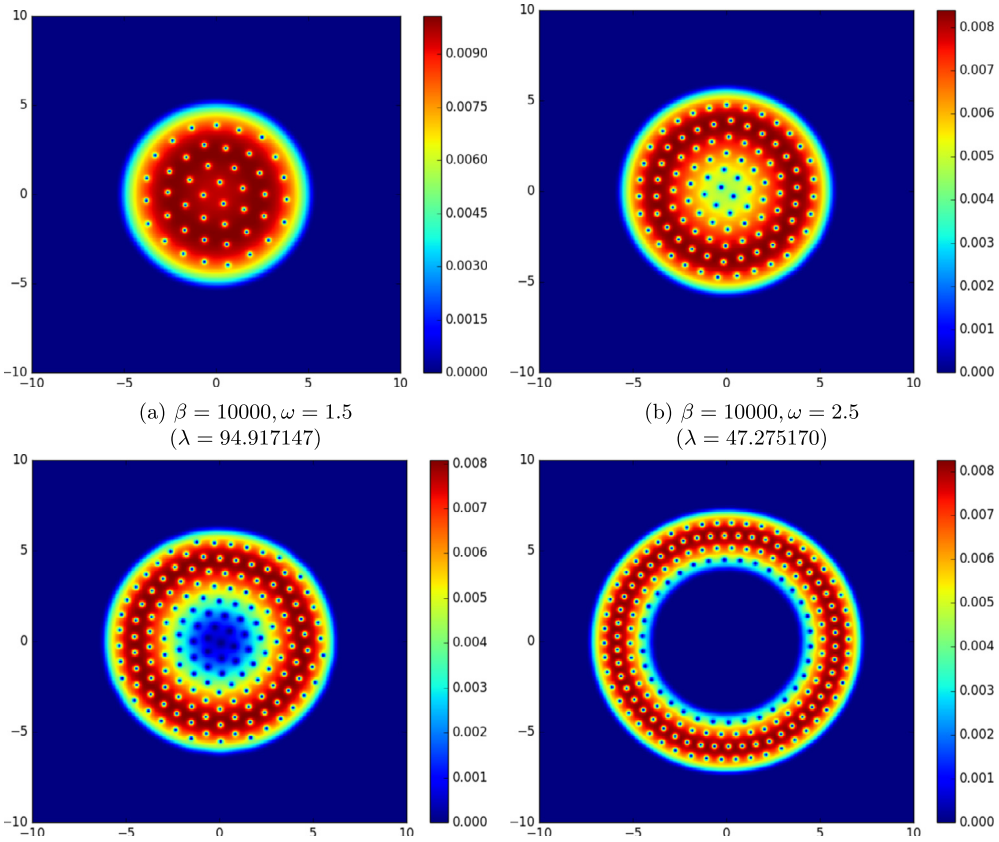
(c) $\beta = 1000, \omega = 2.5$ ($\lambda = -17.158286$)

(d) $\beta = 1000, \omega = 5$ ($\lambda = -515.354718$)

Fig. 14. Contour plots of the density function $|\phi_g|^2$, for $\beta = 1000$ (procs = 32).

Table 1
CPUs time (seconds) to compute the ground states of the GPE for various values of ω and β (procs = 32).

β	$\omega = 1$	1.5	2	2.5	3	3.5	4	4.5
1000	57.79	53.48	91.28	123.36	132.23	173.64	158.41	357.56
5000	97.66	387.07	352.27	685.75	329.34	254.04	253.72	4823.42
10000	489.68	1104.76	1279.08	562.53	960.99	1170.01	3164.85	5603.2



(a) $\beta = 10000, \omega = 1.5$ ($\lambda = 94.917147$)

(b) $\beta = 10000, \omega = 2.5$ ($\lambda = 47.275170$)

(c) $\beta = 10000, \omega = 3$ ($\lambda = 7.297448$)

(d) $\beta = 10000, \omega = 4$ ($\lambda = -141.707175$)

Fig. 15. Contour plots of the density function $|\phi_g|^2$, for $\beta = 10000$ (procs = 32).

Table 2
2D scalability tests.

	# it	time (s)	speedup
1	951	1219.50	1.00
2	888	605.14	2.02
4	891	310.98	3.92
8	886	163.42	7.46
16	914	96.76	12.6
32	876	57.79	21.1

(a) $\beta = 1000, \omega = 1$

	# it	time (s)	speedup
1	1390	1768.86	1.00
2	1411	992.01	1.78
4	1362	488.91	3.62
8	1378	255.94	6.91
16	1347	146.16	12.10
32	1377	91.28	19.38

(b) $\beta = 1000, \omega = 2$

	# it	time (s)	speedup
1	1518	1965.75	1.00
2	1888	1312.84	1.50
4	1805	650.94	3.02
8	1850	344.82	5.70
16	1791	192.15	10.23
32	1839	123.36	15.94

(c) $\beta = 1000, \omega = 2.5$

	# it	time (s)	speedup
1	7746	9976.62	1.00
2	6434	4455.09	2.24
4	7298	2631.25	3.79
8	7246	1355.25	7.36
16	7292	796.05	12.53
32	7387	498.40	20.02

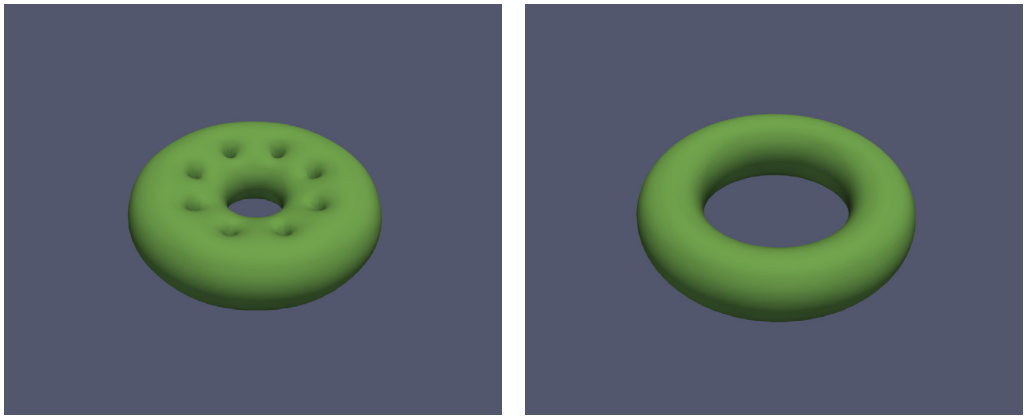
(d) $\beta = 1000, \omega = 5$

Table 3
Ground states of the GPE for $\beta = 10000, \omega = 5, M = 2048$.

	# it	time		speedup
32	18105	10971.45 s	3 hrs	1.0
128	18560	3973.70 s	66 min	2.76

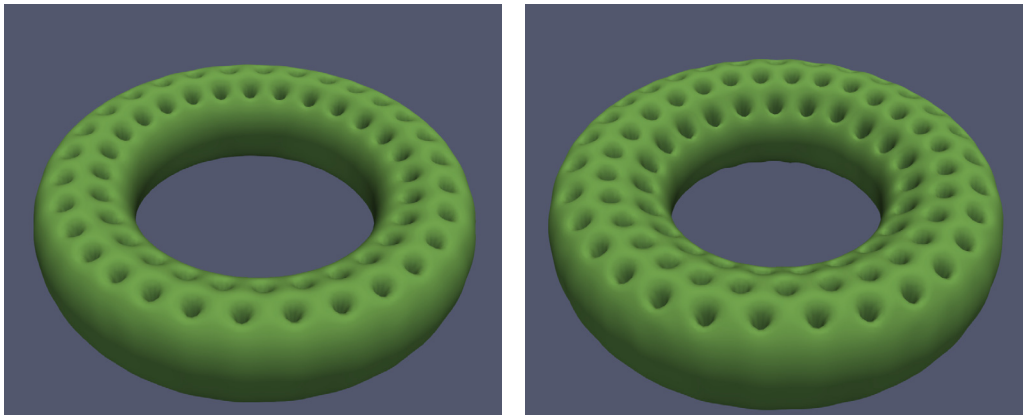
6.3. Numerical results in 3D

We solve now various 3D problems. The computational domain is $[-8, 8]^3$ and the tolerance of the stopping criterion is set to $\varepsilon = 10^{-12}$. Fig. 16 presents the results of four experiments for the mesh size $h = \frac{1}{8}$ ($M = 128$). 3D simulations allow to visualize the torus shape of the BEC as well as the vortices lines. The scalability results presented in Table 4 are again consistent with the scalability of the 3D FFT algorithm and the cost of parallel transpose algorithms. Running on 32 cores can be more than 5 times faster than on 4 cores but the speedup is reduced when the number of iterations increases unfavorably. According to our experiments, this instability in the number of iterations occurs when the grid resolution is not fine enough. Fig. 17 and Table 5 present the results of the same experiments on a finer grid with $h = \frac{1}{16}$ ($M = 256$) for 32 and up to 256 cores. By refining the grids, each iteration is more costly in computational time and the



(a) $\beta = 100, \omega = 1.4$ ($\lambda = 1.878072$)

(b) $\beta = 100, \omega = 1.8$ ($\lambda = -4.106049$)



(c) $\beta = 5000, \omega = 3$ ($\lambda = -40.235518$)

(d) $\beta = 10000, \omega = 3$ ($\lambda = -27.994620$)

Fig. 16. Isosurface $|\phi_g|^2 = 10^{-3}$.

Table 4
3D scalability tests ($M = 128$). The computational domain is $[-8, 8]^3$, $h = \frac{1}{8}$ ($M = 128$), $\varepsilon = 10^{-12}$.

	# it	time (s)	speedup
4	753	1479.90	1.00
8	756	780.97	1.89
16	755	437.01	3.39
32	755	264.64	5.59

(a) $\beta = 100, \omega = 1.4$

	# it	time (s)	speedup
4	514	1057.14	1.00
8	515	537.19	1.97
16	515	292.62	3.61
32	517	177.69	5.95

(b) $\beta = 100, \omega = 1.8$

	# it	time (s)	speedup
4	3632	7019.09	1.00
8	3713	3801.36	1.85
16	3907	2263.74	3.1
32	4257	2409.29	2.91

(c) $\beta = 5000, \omega = 3$

	# it	time (s)	speedup
4	6512	13091.23	1.00
8	6602	6843.93	1.91
16	7450	4352.26	3.01
32	7656	3411.37	3.84

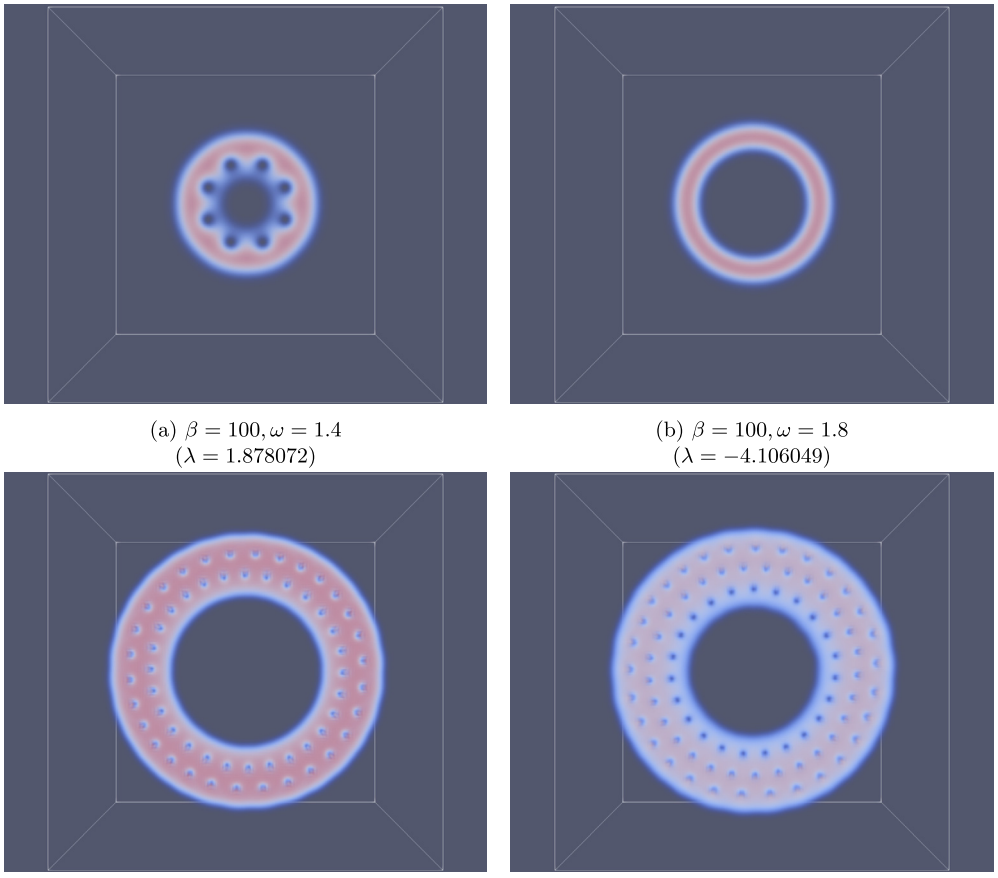
(d) $\beta = 10000, \omega = 3$

resolution also takes more iterations. The computations become rapidly expensive. For example, on this finer grid, more than 3 hours are needed to solve the $\beta = 10000, \omega = 3$ test case on 256 processors.

7. Conclusion

In this paper, we presented BEC2HPC which is a parallel solver for computing the stationary states of the rotating Gross-Pitaevskii equation for the modeling of 2D/3D Bose-Einstein condensates. The scheme implemented in BEC2HPC is based on a preconditioned conjugate gradient for the minimization of the energy functional under normalization constraint, combined with a pseudo-spectral approximation scheme in space (using FFT). This leads to an efficient and robust code for complex problems, that can also be used for problems related to the nonlinear Schrödinger equation. After a presentation of the implementation aspects, we explain how to use the code on a first 2D example. More complicate 2D and 3D test cases are presented next to illustrate some specific coding aspects of the code and to show the scalability of the code for larger problems.

Future developments of BEC2HPC concern the possibility of simulating the dynamics of the rotating GPE by various schemes, the extension to systems of GPE (stationary states and dynamics) and the possibility to simulate nonlocal nonlinear effects like for example for the case of dipole-dipole interactions.



(a) $\beta = 100, \omega = 1.4$ ($\lambda = 1.878072$)

(b) $\beta = 100, \omega = 1.8$ ($\lambda = -4.106049$)

(c) $\beta = 5000, \omega = 3$ ($\lambda = -40.235518$)

(d) $\beta = 10000, \omega = 3$ ($\lambda = -27.994620$)

Fig. 17. 3D color map of $|\phi_g|^2$.

Table 5
3D scalability tests ($M = 256$). Domain is $[-8, 8]^3$, $h = \frac{1}{16}$ ($M = 256$), $\varepsilon = 10^{-12}$.

	# it	time (s)	speedup
32	1609	4541.32	1.00
64	1617	2731.90	1.66
128	1611	2173.89	2.09
256	1611	839.60	5.41

(a) $\beta = 100, \omega = 1.4$

	# it	time (s)	speedup
32	1986	5404.94	1.00
64	2025	3300.06	1.64
128	2028	2366.42	2.28
256	2035	1215.22	4.45

(b) $\beta = 100, \omega = 1.8$

	# it	time (s)	speedup
32	4071	11158.37	1.00
64	4266	7170.47	1.56
128	4330	5832.11	1.91
256	4340	2335.89	4.78

(c) $\beta = 5000, \omega = 3$

	# it	time (s)	speedup
256	22075	12153.14	1.0

(d) $\beta = 10000, \omega = 3$

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors acknowledge the support from the Inria associate team BEC2HPC (Bose-Einstein Condensates: Computation and HPC simulation (https://team.inria.fr/bec2hpc/)). Q. Tang also acknowledges the support from the National Natural Science Foundation of China (No. 11971335). Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

References

[1] M.H. Anderson, J.R. Ensher, M.R. Matthews, C.E. Wieman, E.A. Cornell, Science 269 (5221) (JUL 14 1995) 198–201.
[2] C.C. Bradley, C.A. Sackett, J.J. Tollett, R.G. Hulet, Phys. Rev. Lett. 75 (9) (AUG 28 1995) 1687–1690.
[3] F. Dalfovo, S. Giorgini, L.P. Pitaevskii, S. Stringari, Rev. Mod. Phys. 71 (3) (APR 1999) 463–512.
[4] K.B. David, M.O. Mewes, M.R. Andrews, N.J. Vandruten, D.S. Durfee, D.M. Kurn, W. Ketterle, Phys. Rev. Lett. 75 (22) (NOV 27 1995) 3969–3973.
[5] T. Byrnes, K. Wen, Y. Yamamoto, Phys. Rev. A 85 (4) (2012).

[6] J.R. Abo-Shaeer, C. Raman, J.M. Vogels, W. Ketterle, Science 292 (5516) (APR 20 2001) 476–479.
[7] V. Bretin, S. Stock, Y. Seurin, J. Dalibard, Phys. Rev. Lett. 92 (5) (FEB 6 2004).
[8] K.W. Madison, F. Chevy, V. Bretin, J. Dalibard, Phys. Rev. Lett. 86 (20) (MAY 14 2001) 4443–4446.
[9] K.W. Madison, F. Chevy, W. Wohlleben, J. Dalibard, Phys. Rev. Lett. 84 (5) (JAN 31 2000) 806–809.
[10] M.R. Matthews, B.P. Anderson, P.C. Haljan, D.S. Hall, C.E. Wieman, E.A. Cornell, Phys. Rev. Lett. 83 (13) (SEP 27 1999) 2498–2501.
[11] C. Raman, J.R. Abo-Shaeer, J.M. Vogels, K. Xu, W. Ketterle, Phys. Rev. Lett. 87 (21) (NOV 19 2001).
[12] C. Yuce, Z. Oztas, J. Phys. B, At. Mol. Opt. Phys. 43 (13) (JUL 14 2010).
[13] W. Bao, Y. Cai, Kinet. Relat. Models 6 (1) (MAR 2013) 1–135.
[14] W. Bao, Y. Cai, H. Wang, J. Comput. Phys. 229 (20) (2010) 7874–7892.
[15] W. Bao, Multiscale Model. Simul. SIAM Interdis. J. 2 (2) (2004) 210–236.
[16] W. Bao, Y. Cai, East Asian J. Appl. Math. 1 (2011) 49–81.
[17] X. Antoine, W. Bao, C. Besse, Comput. Phys. Commun. 184 (12) (2013) 2621–2633.
[18] X. Antoine, R. Duboscq, in: C. Besse, J.C. Garreau (Eds.), Nonlinear Optical and Atomic Systems: at the Interface of Physics and Mathematics, in: Lecture Notes in Mathematics, vol. 2146, 2015, pp. 49–145.
[19] A.L. Fetter, B. Jackson, S. Stringari, Phys. Rev. A 71 (Jan 2005) 013605.
[20] B.-W. Jeng, Y.-S. Wang, C.-S. Chien, Comput. Phys. Commun. 184 (3) (2013) 493–508.
[21] S.K. Adhikari, Phys. Lett. A 265 (JAN 17 2000) 91–96.
[22] X. Antoine, R. Duboscq, J. Comput. Phys. 258 (2014) 509–523.
[23] W. Bao, Q. Du, SIAM J. Sci. Comput. 25 (5) (2004) 1674–1697.
[24] D. Baye, J.M. Sparenberg, Phy. Rev. E 82 (5) (Nov 1 2010).
[25] M.M. Cerimele, M.L. Chiofalo, F. Pistella, S. Succi, M.P. Tosi, Phys. Rev. E 62 (1) (JUL 2000) 1382–1389.
[26] M.L. Chiofalo, S. Succi, M.P. Tosi, Phys. Rev. E 62 (5) (NOV 2000) 7438–7444.
[27] R. Zeng, Y. Zhang, Comput. Phys. Commun. 180 (6) (JUN 2009) 854–860.
[28] X. Antoine, R. Duboscq, Comput. Phys. Commun. 185 (11) (2014) 2969–2991.
[29] C.M. Dion, E. Cances, Comput. Phys. Commun. 177 (10) (NOV 15 2007) 787–798.
[30] Y.-S. Wang, B.-W. Jeng, C.-S. Chien, Commun. Comput. Phys. 13 (2013) 442–460.
[31] W. Bao, W. Tang, J. Comput. Phys. 187 (1) (MAY 1 2003) 230–254.
[32] M. Caliari, A. Ostermann, S. Rainer, M. Thalhammer, J. Comput. Phys. 228 (2) (FEB 1 2009) 349–360.
[33] I. Danaila, F. Hecht, J. Comput. Phys. 229 (19) (SEP 20 2010) 6946–6960.
[34] I. Danaila, P. Kazemi, SIAM J. Sci. Comput. 32 (5) (2010) 2447–2467.
[35] I. Danaila, B. Protas, SIAM J. Sci. Comput. 39 (6) (2017) B1102–B1129.
[36] X. Wu, Z. Wen, W. Bao, J. Sci. Comput. 73 (2017) 303–329.
[37] X. Antoine, A. Levitt, Q. Tang, J. Comput. Phys. 343 (2017) 92–109.
[38] X. Antoine, Q. Tang, Y. Zhang, Commun. Comput. Phys. 24 (4) (2018) 966–988.
[39] X. Antoine, Q. Tang, J. Zhang, Int. J. Comput. Math. 95 (6–7) (2018) 1423–1443.
[40] X. Antoine, Q. Tang, Y. Zhang, J. Comput. Phys. 325 (2016) 74–97.
[41] R.K. Kumar, L.E. Young-S, D. Vudragović, A. Balaž, P. Muruganandam, Comput. Phys. Commun. 195 (9) (2015) 117–128.
[42] P. Muruganandam, S.K. Adhikari, Comput. Phys. Commun. 180 (10) (2009) 1888–1912.
[43] D. Vudragović, I. Vidanović, A. Balaž, P. Muruganandam, S.K. Adhikari, Comput. Phys. Commun. 183 (9) (2012) 2021–2025.
[44] V. Lončar, L.E. Young-S, P. Muruganandam, S.K. Adhikari, A. Balaž, Comput. Phys. Commun. 209 (2016) 190–196.
[45] B. Satarič, V. Slavnič, A. Balač, A. Belič, P. Muruganandam, S.K. Adhikari, Comput. Phys. Commun. 200 (2016) 411–417.
[46] L.E. Young-S, P. Muruganandam, S.K. Adhikari, V. Lončar, D. Vudragović, A. Balaž, Comput. Phys. Commun. 220 (2017) 503–506.
[47] R. Kishor Kumar, V. Lončar, P. Muruganandam, S.K. Adhikari, A. Balaž, Comput. Phys. Commun. 240 (2019) 74–82.
[48] U. Hohenester, Comput. Phys. Commun. 185 (1) (2014) 194–216.
[49] R.M. Caplan, Comput. Phys. Commun. 184 (4) (2013) 1250–1271.
[50] Z. Marojević, E. Göklö, C. Lämmerzahl, Comput. Phys. Commun. 202 (2016) 216–232.
[51] G. Vergez, I. Danaila, S. Auliac, F. Hecht, Comput. Phys. Commun. 209 (2016) 144–162.
[52] X. Antoine, R. Duboscq, Comput. Phys. Commun. 193 (2015) 95–117.
[53] W. Bao, S. Jiang, Q. Tang, Y. Zhang, J. Comput. Phys. 296 (2015) 72–89.
[54] P.-A. Absil, R. Mahony, R. Sepulchre, Optimization Algorithms on Matrix Manifolds, Princeton University Press, 2009.
[55] A. Edelman, T.A. Arias, S.T. Smith, SIAM J. Matrix Anal. Appl. 20 (2) (1998) 303–353.
[56] M. Frigo, S.G. Johnson, Proc. IEEE 93 (2) (2005) 216–231, Special issue on "Program Generation, Optimization, and Platform Adaptation".
[57] M. Pippig, SIAM J. Sci. Comput. 35 (3) (2013) C213–C236.
[58] D. Pekurovsky, P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions, CoRR, arXiv:1905.02803 [abs], 2019.
[59] N. Li, S. Laizet, 2DECOMP & FFT - a highly scalable 2D decomposition library and FFT interface, 2010.
[60] The Epetra Project Team, The Epetra Project website, https://trilinos.github.io/epetra.html, 2020.
[61] HashiCorp, Vagrant, https://www.vagrantup.com/, 2010.
[62] HDF Group, et al., HDF5 users guide, http://www.hdfgroup.org/HDF5, 2012.
[63] J. Ahrens, B. Geveci, C. Law, in: The Visualization Handbook, 2005, p. 717.
[64] Y.-S. Wang, C.-S. Chien, J. Comput. Appl. Math. 235 (8) (2011) 2740–2757.